Architecture of Enterprise Applications XI Architectural Patterns – DB Access

Haopeng Chen

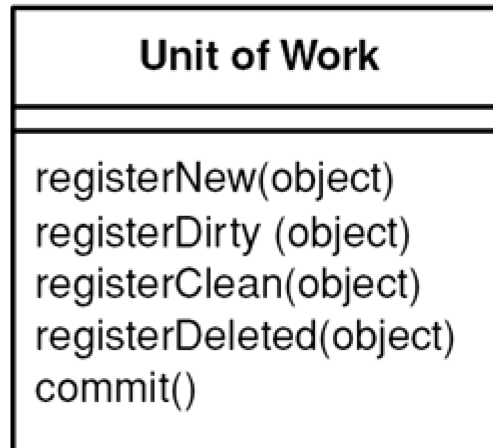
RE*liable,* ***IN****telligent and* ***S****calable Systems Group (****REINS****)*
Shanghai Jiao Tong University
Shanghai, China
e-mail: chen-hp@sjtu.edu.cn

Agenda

- Database Accessing

Database Accessing - Unit of Work

- ORM
  - How to maintain the objects affected by a business transaction and coordinate the writing out of changes and the resolution of concurrency problems.



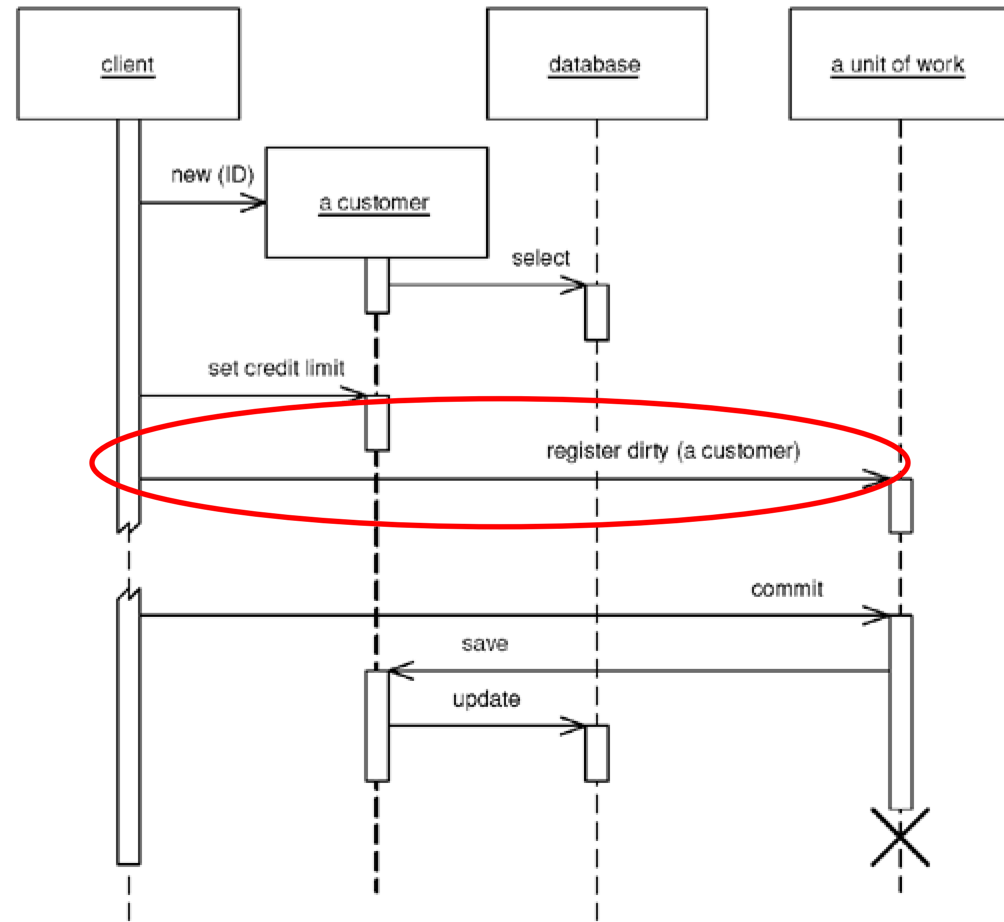
  - A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.

Unit of Work-How It Works

- The obvious things that cause you to deal with the database are changes: new object created and existing ones updated or deleted.
  - Unit of Work is an object that keeps track of these things.
  - As soon as you start doing something that may affect a database, you create a Unit of Work to keep track of the changes.
- The key thing about Unit of Work is that, when it comes time to commit, the Unit of Work decides what to do.
  - Of course for this to work the Unit of Work needs to know what objects it should keep track of.
  - You can do this either by the caller doing it or by getting the object to tell the Unit of Work.

Unit of Work-How It Works

- With caller registration, the user of an object has to remember to register the object with the Unit of Work for changes.
- Any objects that aren't registered won't be written out on commit.
- Although this allows forgetfulness to cause trouble, it does give flexibility in allowing people to make in-memory changes that they don't want written out.

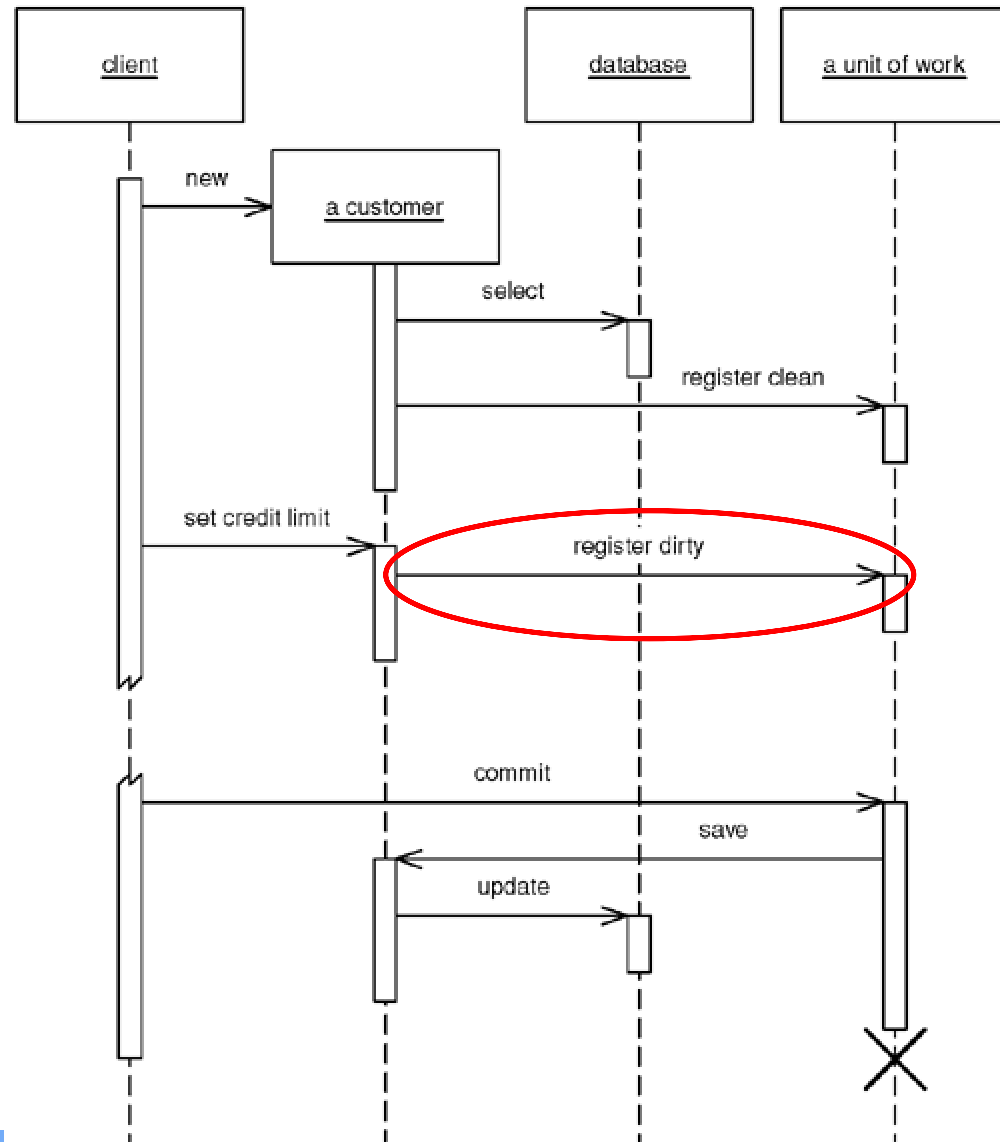


Unit of Work-How It Works

REin
REliable, INtelligent & Scalable Systems

- With object registration, the onus is removed from the caller.
- The usual trick here is to place registration methods in object methods.
  - Loading an object from the database registers the object as clean;
  - the setting methods register the object as dirty.

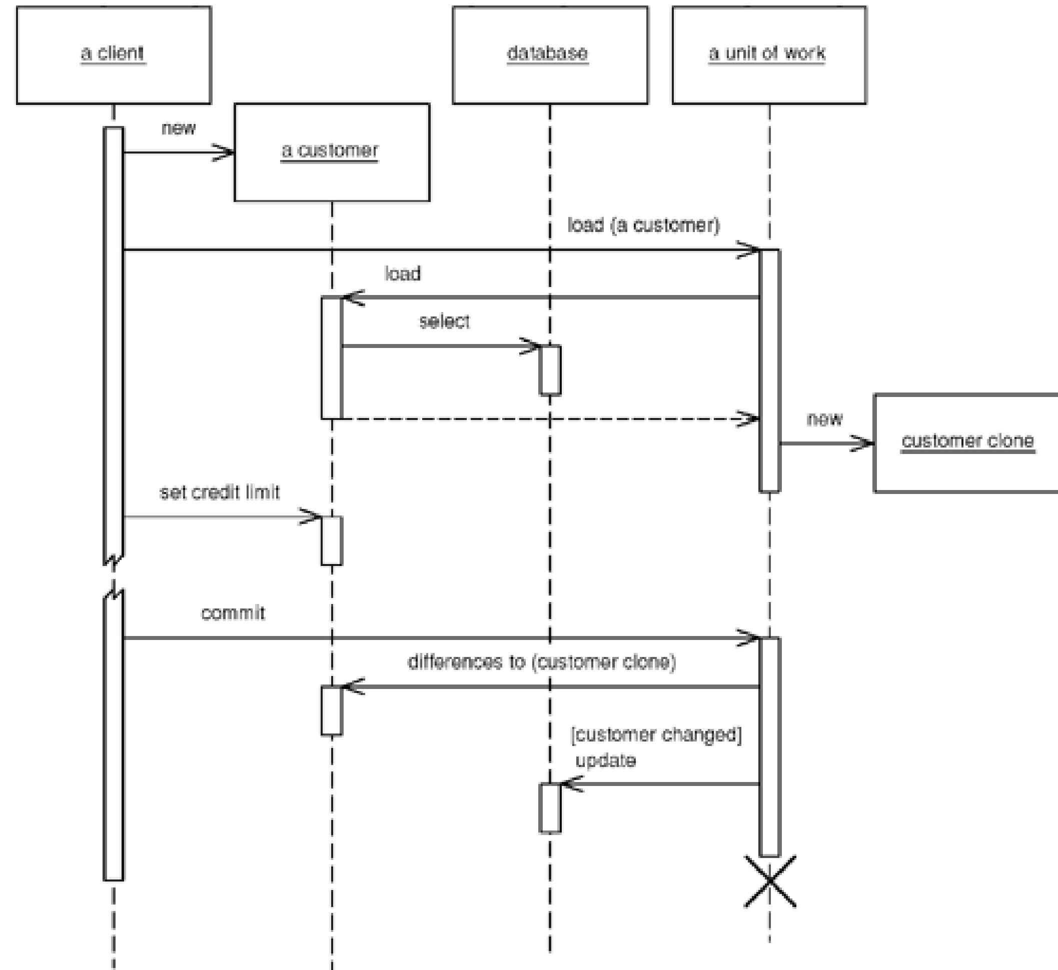
- For this scheme to work the Unit of Work needs either to be passed to the object or to be in a well-known place.



Unit of Work-How It Works

REliable, INtelligent & Scalable Systems

- Another technique is unit of work controller, which the TOPLink product uses. Here the Unit of Work handles all reads from the database and registers clean objects whenever they're read.
  - Rather than marking objects as dirty the Unit of Work takes a copy at read time and then compares the object at commit time.
  - Although this adds overhead to the commit process, it allows a selective update of only those fields that were actually changed; it also avoids registration calls in the domain objects.



Unit of Work-When to Use It

- The fundamental problem that Unit of Work deals with is keeping track of the various objects you've manipulated so that you know which ones you need to consider to synchronize your in-memory data with the database.

- The great strength of Unit of Work is that it keeps all this information in one place. Once you have it working for you, you don't have to remember to do much in order to keep track of your changes.

Unit of Work-Example

- Here's a Unit of Work that can track all changes for a given business transaction and then commit them to the database when instructed to do so.

- To store the change set we use three lists: new, dirty, and removed domain objects.

```
class UnitOfWork...
 private List newObjects = new ArrayList();
 private List dirtyObjects = new ArrayList();
 private List removedObjects = new ArrayList();
```

Unit of Work-Example

- The registration methods maintain the state of these lists. They must perform basic assertions such as checking that an ID isn't null or that a dirty object isn't being registered as new.

```
class UnitOfWork...
 public void registerNew(DomainObject obj) {
  Assert.notNull("id not null", obj.getId());
  Assert.isTrue("object not dirty", !dirtyObjects.contains(obj));
  Assert.isTrue("object not removed", !removedObjects.contains(obj));
  Assert.isTrue("object not already registered new", !newObjects.contains(obj));
  newObjects.add(obj);
 }

 public void registerDirty(DomainObject obj) {
  Assert.notNull("id not null", obj.getId());
  Assert.isTrue("object not removed", !removedObjects.contains(obj));
  if (!dirtyObjects.contains(obj) && !newObjects.contains(obj)) {
   dirtyObjects.add(obj);
  }
 }
```

Unit of Work-Example

REliable, INtelligent & Scalable Systems

```
public void registerRemoved(DomainObject obj) {
 Assert.notNull("id not null", obj.getId());
 if (newObjects.remove(obj))
  return;
 dirtyObjects.remove(obj);
 if (!removedObjects.contains(obj)) {
  removedObjects.add(obj);
 }
}

public void registerClean(DomainObject obj) {
 Assert.notNull("id not null", obj.getId());
}
```

Unit of Work-Example

REliable, INtelligent & Scalable Systems

- commit() will locate the Data Mapper for each object and invoke the appropriate mapping method. updateDirty() and deleteRemoved() aren't shown, but they would behave like insertNew(), which is as expected.

```
class UnitOfWork...
 public void commit() {
  insertNew();
  updateDirty();
  deleteRemoved();
 }
 private void insertNew() {
  for (Iterator objects = newObjects.iterator(); objects.hasNext();) {
   DomainObject obj = (DomainObject) objects.next();
   MapperRegistry.getMapper(obj.getClass()).insert(obj);
  }
}
```

Unit of Work-Example

REliable, INtelligent & Scalable Systems

- As each business transaction executes within a single thread we can associate the Unit of Work with the currently executing thread using the java.lang.ThreadLocal class.

```
class UnitOfWork...
 private static ThreadLocal current = new ThreadLocal();
 public static void newCurrent() {
  setCurrent(new UnitOfWork());
 }
 public static void setCurrent(UnitOfWork uow) {
  current.set(uow);
 }
 public static UnitOfWork getCurrent() {
  return (UnitOfWork) current.get();
 }
```

Unit of Work-Example

- Now we can now give our abstract domain object the marking methods to register itself with the current Unit of Work.

```
class DomainObject...
 protected void markNew() {
  UnitOfWork.getCurrent().registerNew(this);
 }
 protected void markClean() {
  UnitOfWork.getCurrent().registerClean(this);
 }
 protected void markDirty() {
  UnitOfWork.getCurrent().registerDirty(this);
 }
 protected void markRemoved() {
  UnitOfWork.getCurrent().registerRemoved(this);
}
```

Unit of Work-Example

REin
REliable, INtelligent & Scalable Systems

- Concrete domain objects need to remember to mark themselves new and dirty where appropriate.

```
class Album...
 public static Album create(String name) {
  Album obj = new Album(IdGenerator.nextId(), name);
  obj.markNew();
  return obj;
 }
 public void setTitle(String title) {
   this.title = title;
   markDirty();
}
```

Unit of Work-Example

REin

REliable, INtelligent & Scalable Systems

- The final piece is to register and commit the Unit of Work where appropriate. This can be done either explicitly or implicitly. Here's what explicit Unit of Work management looks like:

```
class EditAlbumScript...
 public static void updateTitle(Long albumId, String title) {
  UnitOfWork.newCurrent();
  Mapper mapper = MapperRegistry.getMapper(Album.class);
  Album album = (Album) mapper.find(albumId);
  album.setTitle(title);
  UnitOfWork.getCurrent().commit();
}
```

Unit of Work-Example

REliable, INtelligent & Scalable Systems

- Here's a servlet that registers and commits the Unit of Work for its concrete subtypes. Subtypes will implement handleGet() rather than override doGet(). Any code executing within handleGet() will have a Unit of Work with which to work.

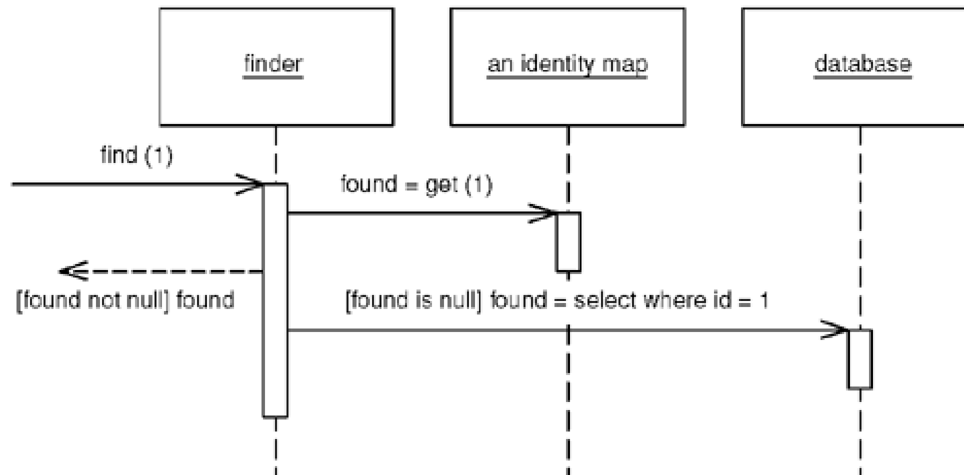
```
class UnitOfWorkServlet...
 final protected void doGet(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException {

   try {
     UnitOfWork.newCurrent();
     handleGet(request, response);
     UnitOfWork.getCurrent().commit();
   } finally {
     UnitOfWork.setCurrent(null);
   }
 }

 abstract void handleGet(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException;
```

Database Accessing

- ORM
  - How to guarantee the same data will be loaded only once?
    - Identity Map: Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them.



    - An Identity Map keeps a record of all objects that have been read from the database in a single business transaction. Whenever you want an object, you check the Identity Map first to see if you already have it.

Identity Map-How It Works

REin
REliable, INtelligent & Scalable Systems

- The basic idea behind the Identity Map is to have a series of maps containing objects that have been pulled from the database.
  - In a simple case, with an isomorphic schema, you'll have one map per database table. When you load an object from the database, you first check the map.
    - If there's an object in it that corresponds to the one you're loading, you return it.
    - If not, you go to the database, putting the objects into the map for future reference as you load them.

- Choice of Keys
  - The obvious choice is the primary key of the corresponding database table.
- Explicit or Generic
  - findPerson(1) or find("Person", 1).
- How Many
  - one mapper class or one map for the whole session.

- Where to Put Them
  - You need to ensure that each session gets it's own instance that's isolated from any other session's instance.
  - Thus, you need to put the Identity Map on a session-specific object.

Identity Map-When to Use It

- In general you use an Identity Map to manage any object brought from a database and modified.
  - The key reason is that you don't want a situation where two in-memory objects correspond to a single database record-you might modify the two records inconsistently and thus confuse the database mapping.
- Another value in Identity Map is that it acts as a cache for database reads, which means that you can avoid going to the database each time you need some data.
- You may not need an Identity Map for immutable objects.
- Identity Map helps avoid update conflicts within a single session, but it doesn't do anything to handle conflicts that cross sessions.

Identity Map-Example

REin
REliable, INtelligent & Scalable Systems

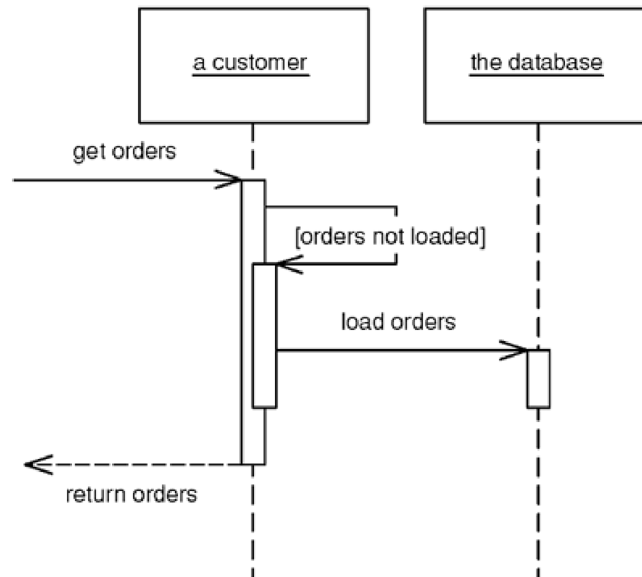
- For each Identity Map we have a map field and accessors.

```
private Map people = new HashMap();
public static void addPerson(Person arg) {
  soleInstance.people.put(arg.getID(), arg);
}
public static Person getPerson(Long key) {
  return (Person) soleInstance.people.get(key);
}
public static Person getPerson(long key) {
  return getPerson(new Long(key));
}
```

- One of the annoyances of Java is the fact that long isn't an object so you can't use it as an index for a map.

Database Accessing

- ORM
  - Lazy load vs. Eager load?
    - Lazy load An object that doesn't contain all of the data you need but knows how to get it.



    - A Lazy Load leaves a marker in the object structure so that if the data is needed it can be loaded only when it is used. As many people know, if you're lazy about doing things you'll win when it turns out you don't need to do them at all.

Lazy Load-How It Works

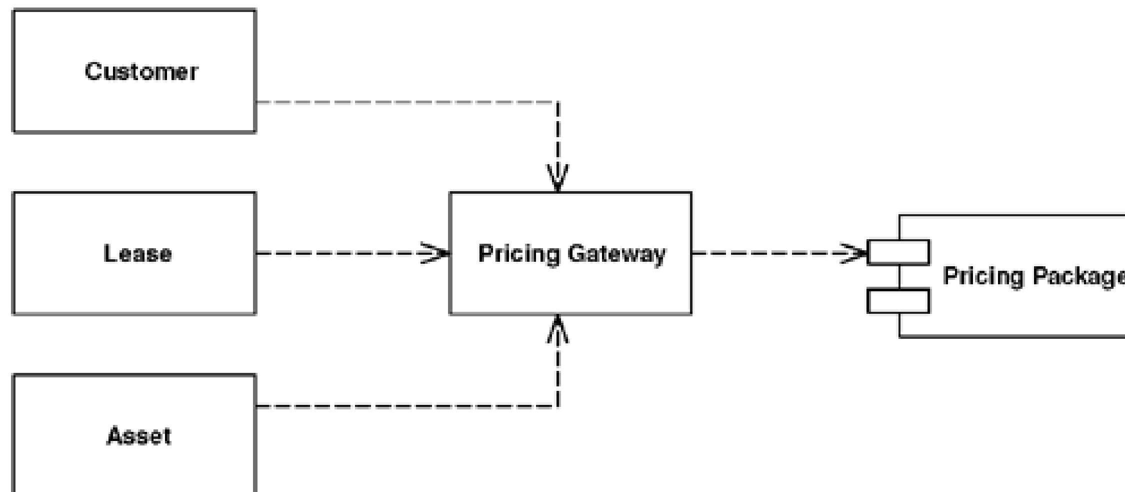
- Inheritance often poses a problem with Lazy Load.
  - If you're going to use ghosts, you'll need to know what type of ghost to create, which you often can't tell without loading the thing properly.
  - Virtual proxies can suffer from the same problem in statically typed languages.
- Another danger with Lazy Load is that it can easily cause more database accesses than you need.
  - A good example of this ripple loading is if you fill a collection with Lazy Loads and then look at them one at a time.
  - One way to avoid it is never to have a collection of Lazy Loads but, rather make the collection itself a Lazy Load and, when you load it, load all the contents.
- In theory you might want a range of different degrees of laziness, but in practice you really need only two: a complete load and enough of a load for identification purposes in a list.

Lazy Load-When to Use It

- Deciding when to use Lazy Load is all about deciding how much you want to pull back from the database as you load an object, and how many database calls that will require.
  - It's usually pointless to use Lazy Load on a field that's stored in the same row as the rest of the object, because most of the time it doesn't cost any more to bring back extra data in a call, even if the data field is quite large.
  - That means it's usually only worth considering Lazy Load if the field requires an extra database call to access.
- In performance terms it's about deciding when you want to take the hit of bringing back the data.
  - Often it's a good idea to bring everything you'll need in one call so you have it in place, particularly if it corresponds to a single interaction with a UI.
  - The best time to use Lazy Load is when it involves an extra call and the data you're calling isn't used when the main object is used.

Database Accessing

- How to access non-relational DBMS?
  - Requirement:
    - We need to access a file-based system to get the historical data about students.
  - GateWay
    - An object that encapsulates access to an external system or resource.



Database Designing

- Cluster or not? And how?
  - Clustering is a way to improve the processing ability of system.
  - Load balancing and communication in cluster do a negative effect on performance.
  - Horizontal clustering vs. Vertical clustering
  - The relationship between scalability and the scale of cluster is NOT linear.

References

- Martin Fowler's Patterns of Enterprise Application Architecture

Thank You!